HPC-Reuse: efficient process creation for running MPI and Hadoop MapReduce on supercomputers

Thanh-Chung Dao
Dept. of Creative Informatics
The University of Tokyo
Email: chung@csg.ci.i.u-tokyo.ac.jp

Shigeru Chiba
Dept. of Creative Informatics
The University of Tokyo
Email: chiba@acm.org

***Abstract*—Hadoop and Spark analytics are used widely for large-scale data processing on commodity clusters. It is better choice to run them on supercomputers in aspects of productivity and maturity rather than developing new frameworks from scratch. YARN, a key component of Hadoop, is responsible for resource management. YARN adopts dynamic management for job execution and scheduling. We identify three Ds (3D) *dynamic* characteristics from YARN-like management: on-Demand (processes created during job execution), Diverse job, and Detailed (fine-grained allocation). The dynamic management does not fit into typical resource managers on supercomputers, for example PBS, that are identified having three Ss (3S) *static* characteristics: Stationary (no newly created process during execution), Single job, and Shallow (coarse-grained allocation). In this paper, we propose HPC-Reuse located between YARN-like and PBS-like resource managers in order to provide better support of dynamic management. HPC-Reuse helps avoid process creation, such as MPI-Spawn, and enable MPI communication over Hadoop processes. Our experimental results show that HPC-Reuse can reduce execution time of iterative PageRank by 26%.**

I. Introduction

Supercomputing has focused mainly on compute-intensive applications, but data-intensive workloads are emerging as supercomputing problems. Hadoop [1] and Spark [2] analytics are used widely for large-scale data processing on commodity clusters. It is better choice to run them on supercomputers in aspects of productivity and maturity rather than developing new frameworks from scratch. Hadoop and Spark depend on YARN [3] and Mesos [4] resource managers to execute their tasks, whereas most supercomputers use PBS [5] and Slurm [6]. There are some studies on running Hadoop and Spark as typical jobs submitted to the PBS or Slurm's job queue [7] [8] [9]. To the best of our knowledge, however, there is no study carried out to adapt and optimize YARN-like resource managers to ones on supercomputers.

A. YARN-like resource managers

YARN adopts dynamic management for job execution and scheduling in order to increase resource utilization. We identify three Ds (3D) *dynamic* characteristics from YARN-like management as follows:

on-Demand. Execution processes are created on demand. Number of processes and CPU time are subject to change during job execution that depend on size of input and output data.

Diverse. It supports running simultaneously multiple jobs with fine-grained scheduling. Job types are not limited to MapReduce since YARN can host Spark and Storm as well.

Detailed (fine-grained). Resources are managed by number of process containers running on each node based on amount of memory. The resource manager is responsible for scheduling, process executing, and handling failures.

B. Issue of resource management

The dynamic management does not fit into typical resource managers on supercomputers that are identified having the below three Ss (3S) *static* characteristics:

Stationary. It is typical to start all necessary processes at the beginning of job running and there is no newly created process during execution (e.g. MPI programs). Number of processes and CPU time should be specified at the job submission.

Single. It is typical to run jobs one by one on allocated nodes. Simultaneous jobs can affect jobs' performance because resource managers only support coarse-grained manner in which each job must handle resources that it is using.

Shallow (coarse-grained). Resources are often requested by number of nodes and the whole resource on a node is allocated to a user's job. Users decide how many processes run on those nodes, how they are executed, and how to handle failures.

Dynamic management on supercomputers is supported partially and not sufficient to be integrated with resource managers, such as Hadoop YARN. For example, FX10 supercomputer allows to fork a new process but MPI is not available in that process, whereas Tsubame supercomputer provides

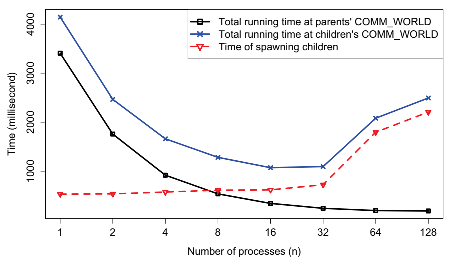


Fig. 1. *MPI application (Prime count) running on parent processes versus spawned child processes on Tsubame supercomputer*

creating dynamically processes by using MPI-Spawn command, but its performance is slow due to a collective operator required. MPI is de facto communication on supercomputers and highly optimized for the associated interconnection. It stands for Message Passing Interface that is used widely on supercomputer environment rather than TCP/IP. MPI-Spawn is a mechanism to spawn new processes on which MPI connection is still kept. FX10 does not allow using MPI-Spawn to spawn new processes on the same node. Figure 1 shows a MPI application, Prime count, running on 32 Tsubame nodes and using OpenMPI 1.6.5. In the first test case, we run the application on the *COMM_WORLD* communicator and measure total running time. In the second one, we spawn child processes from the *COMM_WORLD* communicator and run the same program on those processes. It reveals that the spawning time is relatively long in comparison with total execution time, and it takes 0.5 second to spawn a process.

To provide better support of dynamic management, we propose a virtual layer located between YARN-like and PBS-like resource managers on supercomputers. It helps avoid process creation, such as MPI-Spawn, and enable MPI communication over Hadoop processes. In order to avoid process creation later and satisfy resource specification at the beginning of job running, we create a bunch of processes in advance and then allocate them to the dynamic resource manager when it requests process execution. These processes are cleaned and de-allocated back in order to be used next time when they finish. We use a process pool to implement this mechanism. We name it HPC-Reuse designed specially for Hadoop YARN hosting JVM-based applications, such as MapReduce, Spark, and Storm applications, running on supercomputers.

Compared to the original Hadoop, our experimental results show that HPC-Reuse can reduce execution time of iterative PageRank by 26% on FX10 supercomputer. For Tera-sort running on Tsubame supercomputer, the improvement is minor, but compared to using MPI-Spawn to create new processes, HPC-Reuse has achieved improvement of 52% on average.

II. HPC-Reuse

In this section, first we describe the idea of reusing applied to YARN and show its design. Technical issues are followed.

A. Idea of Reusing

YARN creates process containers on demand when it receives requests from workloads. *Container* is used as abstraction for resource allocation. It has two main components: *ResourceManager* (RM) and *NodeManager* (NM) running on master and slave nodes, respectively. RM is responsible for allocating resources to workloads and scheduling, and NM is responsible for executing containers and monitoring them on a node. It is typical to have one RM. NM uses the fork-based mechanism to spawn new JVM process containers when it receives task running requests (e.g. MapTasks and ReduceTasks) from a workload. Figure 2 (left) shows the creation flow. First, a shell script containing a MapTask or ReduceTask is prepared in advance. Then, it is invoked to export environment variables

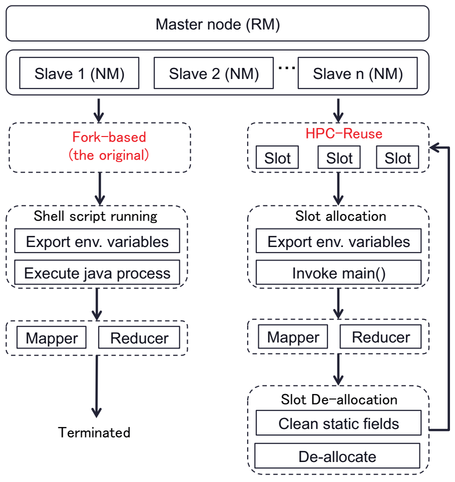


Fig. 2. *Fork-based vs. HPC-Reuse workflow: slots denote JVM processes; RM and NM stand for ResourceManager and NodeManager, respectively.*

and run the process container (called Mapper or Reducer). When the assigned task is completed, the process container is terminated.

To avoid process creation, we keep JVM process containers running without being terminated. A pool of empty JVM process containers is responsible for container allocation and de-allocation to each execution request. Figure 2 (right) illustrates this approach (called HPC-Reuse). When NM receives a container request, it sends that request containing the shell script path to HPC-Reuse. Then, a slot is assigned to run the shell script and invoke the *main()* of the Java program. When the assigned task is completed, that slot is cleaned and returned back to the pool.

B. Process pool

We use process pool approach to implement HPC-Reuse. A pool containing empty JVM processes (namely slots) is created on each node at the beginning when the Hadoop cluster is deployed on the supercomputer. Note that number of slots is fixed and unchanged during job execution. Figure 3 shows the pool architecture on a slave node where a NodeManager is running. Container requests are forwarded from NM to *Pool Manager* instead of calling a process builder (fork-based approach). A flag array is used to store slot status in the pool. We use round-robin scheduling to choose an empty slot. A container request is held until there is a slot available.

When a slot is assigned to a container request, first, environment variables are exported. Then, a new class loader is created to load user classes. Finally, *main()* method is invoked to run the associated task. De-allocation including static field clean-up and slot reseting will be called after the task is completed.

C. Technical issues

In order to make allocation and de-allocation possible and safe, there are two issues: how to load user classes and clean

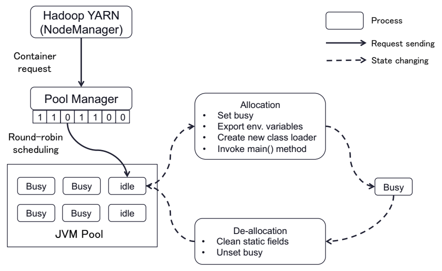


Fig. 3. *Pool architecture in a node: one represents an occupied state and zero represents an empty state*

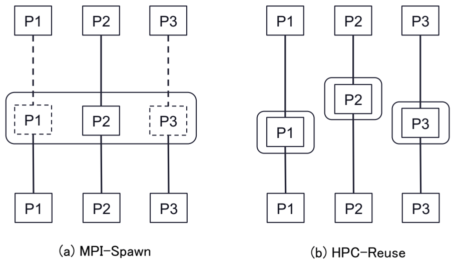


Fig. 4. *MPI-Spawn vs. HPC-Reuse in process creation: rounded rectangles represent spawning call; rectangles denote processes; dotted rectangles show waiting states.*

a slot. Note that these issues are problematic for JVM-based applications running on YARN, such as MapReduce, Spark, and Storm.

1) Class loading: We use a new class loader to load user classes when a user submits a MapReduce workload to the Hadoop cluster. In the original flow of process creation (fork-based approach), *CLASSPATH* containing user classes is exported before container execution, so the container can find and load such classes. In HPC-Reuse flow, however, slots in the pool are started before the *CLASSPATH* is exported and thus the user classes are not found.

At the step of slot allocation in HPC Reuse workflow, a new class loader is created to load the user classes before invoking its *main()* method. Even if the user submits the same class and package that exists in the previous workload, an error will not happen. A class loader is created newly for each workload. However, note that we do not reload all Hadoop classes, and only the user classes are reloaded. That helps reduce class loading time and exploit compilation technology in JVM.

2) Clean-up: HPC-Reuse may have a problem of security due to reusing the JVM in which static fields for the previous workload are already set. For example, when the container is executed, loginUser static field is initialized. That field is kept unchanged whenever its value is not *null*. Therefore, the same static field is used for other users. Note that UserGroupInformation class, which contains loginUser, is not reloaded for each workload since it is already loaded by the system class loader. Although on the supercomputer, Hadoop cluster is used by only one user at a time, but clean-up of all static fields is necessary in general. However, at the current implementation, we just do a simple clean-up by reseting only static fields containing user information and workload configuration.

D. Performance benefit

As mentioned in the motivation section, MPI-Spawn is a mechanism to spawn a new MPI process, but its performance is slow. Note that nested MPI jobs are not allowed on most of implementation of MPI on supercomputers. Figure 4 illustrates MPI-Spawn's collective operation: rounded rectangles represent spawning call; rectangles denote processes; dotted rectangles show waiting states. When spawning is invoked on the *COMM_WORLD* communicator at a certain process, it requires all other processes must call spawning with the same parameters (Fig. 4(a): P1 and P3 must stop their execution while P2 is calling). Each spawning call must be serialized. Conversely, HPC-Reuse supports multiple threads and does not use MPI-Spawn, so each process can request creating as many new processes as needed at any time (Fig. 4(b)). HPC-Reuse keeps those processes running without terminating, so MPI connection is always available during job execution time.

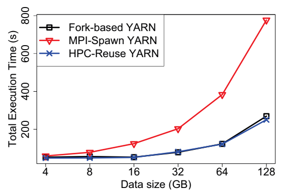
III. Experimental Evaluation

Our experiments are conducted on the 33 Tsubame nodes or 33 FX10 compute nodes. A Tsubame node (thin type) is equipped with Intel Xeon X5670 2.93GHz processor (12 cores) and 54GB main memory. All compute nodes are connected using the Infiniband network (Grid Director 4700). Its maximum throughput is 80 Gbps. Each node has 120GB of local SSD storage and parallel shared disks are also provided. Each FX10 node is equipped with SPARC64 IXfx 1.848 GHz processor (16 cores) and 32GB main memory. Computing nodes are connected with each other through Tofu interconnection [10]. FX10 does not have a local disk for each computing node, conversely shared disks used.

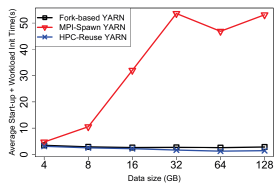
HPC-Reuse can be integrated with any Hadoop version *2.x*, but we use Hadoop v2.2.0 (a stable version) in our evaluation. We use OpenJDK 7 and OpenMPI 1.6.5 on both Tsubame and FX10. In order to use MPI from Java, we use a MPI binding [11] that has been included in OpenMPI 1.7.5 or later. We made a minor modification to integrate this MPI binding with OpenMPI on the Tsubame and FX10. On FX10, experiments are run with the MCA parameter *plm_ple_cpu_affinity = 0* to disable CPU binding to each MPI process.

We use three test cases: fork-based YARN, MPI-Spawn YARN, and HPC-Reuse YARN. The original YARN is called fork-based YARN. We replaced the process fork mechanism of the original YARN with using MPI-Spawn command (at DefaultContainerExecutor class), which we call MPI-Spawn YARN. HPC-Reuse YARN is our proposal.

This experiment is aimed to show performance of HPC-Reuse YARN is as good as the original fork-based YARN and HPC-Reuse approach shortens start-up time in iterative workloads. We run Tera-sort workload of different input size up to 128GB on Tsubame. We calculate the average execution and start-up time for each input size. We run iterative PageRank

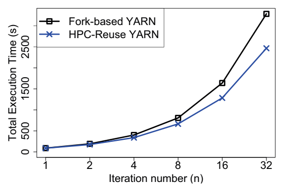


(a) Total execution time

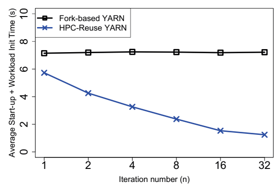


(b) Average start-up time + workload init

Fig. 5. Benefit of HPC-Reuse (Tera-sort on Tsubame)



(a) Total execution time



(b) Average start-up time + workload init

Fig. 6. Benefit of HPC-Reuse (Iterative PageRank on FX10)

on FX10 and measure its total execution and start-up time. Each experiment of a data size or an iteration is run twice. Note that MPI-Spawn YARN does not work on FX10 because it is not allowed to spawn new processes on the same node. For fair comparison, the original shuffle engine (TCP/IP) is used for three test cases. Number of Reducers is set equal to number of Mappers in each experiment.

Figure 5 shows results of Tera-sort running on Tsubame. In Figure 5a, HPC-Reuse outperforms the MPI-Spawn approach when data size is bigger than 32GB and its performance is the same as the fork-based one, even better with 128GB data size. Compared to MPI-Spawn YARN and Fork-based YARN, HPC-Reuse YARN reduces the workload execution times by 52% and 6%, respectively, on average. Regarding start-up time (Figure 5b), it achieves improvement of 82% and 30%, respectively, on average.

Figure 6 shows results of iterative PageRank running on FX10. In Figure 6a, when the number of iterations increases, HPC-Reuse approach shows more reduction, up to 26% at 32 iterations. This is because the more number of iteration is executed, the more JVM process container are created that makes start-up time of Fork-based YARN longer. HPC-Reuse helps eliminate overhead of JVM start-up. In Figure 6b, HPC-Reuse reduces start-up and workload initialization time by 57% on average.

IV. Related work

Resource management. Slurm++ workload manager [12] is an extended implementation from Slurm that aims to support mixture of applications, such as traditional HPC applications (MPI), ensemble runs, and many-task computing at exascale. It uses multiple controllers to manage partitions of compute nodes, but its current design only supports running coarse-grained workloads. Our HPC-Reuse YARN can host various kinds of application in a fine-grained management manner.

JVM Reuse. M3R [13] used X10 language and JVM Reuse to implement HMR engine and run in-memory MapReduce. However, technical issues were not provided, for example class loading and static field clean-up. Also, there was no specific evaluation of JVM Reuse, such as start-up time reduction. In our HPC-Reuse, we provide optimization to use JVM Reuse more efficiently, and its effectiveness on iterative workloads has been evaluated. Moreover, we keep the original HMR engine with minimum changes.

In the Hadoop v1 (without YARN), MapTasks can be executed in a JVM process in sequence (*mapreduce.job.jvm.numtasks*) [14]. However, this JVM process is only used for a single workload, and it will be terminated when MapTasks finish.

V. Conclusion

We have proposed HPC-Reuse to better support dynamic management on supercomputers. It helps avoid process creation mechanism, such as MPI-Spawn, and enable MPI communication over Hadoop processes. Our experimental results show effectiveness of HPC-Reuse with improvement of 26% in iterative PageRank.

References

[1] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.
[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.
[3] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.
[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.
[5] A. Bayucan, R. L. Henderson, J. P. Jones, C. Lesiak, B. Mann, B. Nitzberg, T. Proett, and J. Utley, "Portable batch system openpbs release 2.3 administrator guide," 2000.
[6] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.
[7] S. Krishnan, M. Tatineni, and C. Baru, "myhadoop-hadoop-on-demand on traditional hpc resources," *San Diego Supercomputer Center Technical Report TR-2011-2, University of California, San Diego*, 2011.
[8] M. Gajbe, K. Chadalavada, G. Bauer, and W. Kramer, "Benchmarking and performance studies of mapreduce/hadoop framework on blue waters supercomputer."
[9] T. Baer, P. Peltz, J. Yin, and E. Begoli, "Integrating apache spark into pbs-based hpc environments," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*. ACM, 2015, p. 34.
[10] Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: A 6d mesh/torus interconnect for exascale computers," *Computer*, vol. 11, no. 42, pp. 36–40, 2009.
[11] O. Vega-Gisbert, J. E. Roman, and J. M. Squyres, "Design and implementation of java bindings in open mpi," 2014.
[12] K. Wang, X. Zhou, K. Qiao, M. Lang, B. McClelland, and I. Raicu, "Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015, pp. 219–222. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749249
[13] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta, "M3r: increased performance for in-memory hadoop jobs," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1736–1747, 2012.
[14] R. Stewart and J. Singer, "Comparing fork/join and mapreduce," Citeseer, Tech. Rep., 2012.